PART 1 – SOFTWARE ARCHITECTURE

SIMONE GUGGIARI
ETH ZÜRICH 2016
LEGI # 14-932-867

1. INTRODUCTION

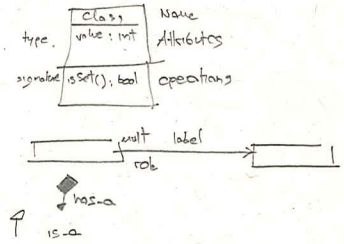
- Software failures
  - exampes of bugs
    - patriot, therac, windows
  - track record
    - failure causes
- Challenges
  - complexity (huge # states + paths)
  - change (deviation, addition)
  - competing objectives (16)
  - constraints (budget, time, staff)
- Solution approaches
  - software design
  - testing
- overview
  - (formal) modeling, patterns
  - functional/structural testing, test case generation

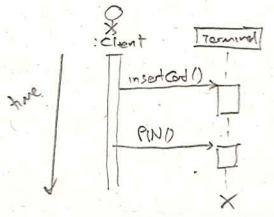
2. REQUIREMENTS ELICITATION

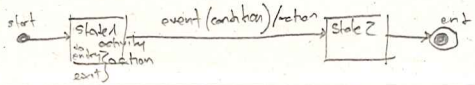
- Activities
  - req. elicitation, design, implementation, validation
- Requirements
  - definition
  - user's view, what
  - functionality, interfaces (functional)
  - performance, attributes, design constraints (non-functional)
  - quality criteria (7)
  - validation
    - reviews
    - prototyping
- Activities - identifying:
  - actors = roles
  - scenario = common case, understandability
  - use case = generalization, completeness
    - event flow, conditions
  - non-functional requirements

3. MODELING AND SPECIFICATIONS

- Mastering complexity
- design =
  - system design (composition of subsystems)
    - components, connectors
  - detailed design
    - choose implementation (DnA)
- Examples
  - map / initialization / list / (reference counting)
- Code documentation
  - design documentation
  - communicate assumptions
  - examples
    - hashmap, imagefile, list
  - source code inefficient
- what to document
  - clients: use, interface
    - constructors, methods, fields, super-types
    - calls, parameters, results, effects, input/output
  - implementors: specific details
    - type of copy, global properties, consistency
    - data structures, algorithms, invariants, evolution in time
    - assumptions
  - what, not how
- How to document
  - comments
  - types and modifiers
  - effect system
  - metadata (@tags)
  - assertions
  - contracts (pre/post)
- Techniques
- INFORMAL MODELS
  - underspecification, modeling
  - views on the design
  - design specification, asbstraction
- UML
  - (static models) = structure
  - Class diagram
    - state, behavior
    - instances
    - associations, connections
    - (multiplicity, label, navigability)
    - composition (non-sharing ownership) "has-a"
    - generalization (inheritance, specialization) "is-a"
  - (dynamic models) = behavior
  - Sequence diagram
    - interaction, collaboration
    - message, data flow
    - creation, destruction
    - parallel, alternative execution
  - State, activity diagram
    - state dependent behavior
    - finite state machine







    - events, actions, activities
  - Use case diagram (not seen) – requirements
  - Implementation diadgram (not seen) – code dependencies
    - component / deployment
  - Contracts
  - OCL – object constraint language
    - (invariants, pre/post)

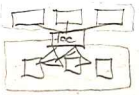
- self
- attributes
- role names
@pre

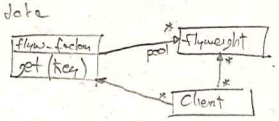
- Mapping models to code
  - model-driven deployment
  - advantages
  - problems
    - abstraction mismatch
    - incomplete/informal specifications
  - mapping classes to signature code
- FORMAL MODELS
  - mathematical precision
- ALLOY
  - set theory
  - collection of constraints
  - ***** SKIPPED CODE *****
  - dynamic models
    - state/time
  - abstract machine idiom (AMI)
    - facts = properties maintained by operations
    - see examples
    - initialization / preservation
    - temporal invariants (next, lt, lte)
    - Traces (+ properties)
- Analyzing models
  - Consistency (satisfiable) $\exists s.\ C(s) \wedge F(s)$
    - run
  - Validity (always true) $\forall s.\ C(s) \Rightarrow F(s)$
    - check
    - use invalidity
  - Scope (otherwise undecidable)
  - Translation into bool formula
  - SAT solver
  - unsat response
  - small model hypothesis

util/ordering [Class]
↑ constraint ↑ formula to test

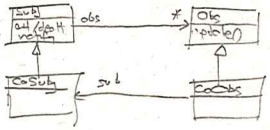
4. MODULARITY

- COUPLING definition
  - summary (trade-offs)
- Data coupling
  - Representation exposure
  - shared data structures
  - 1. restrict access to data (narrow interface)
    - no leaking/capturing, clone
    - Façade pattern
  - 2. shared data immutable
    - Flyweight pattern → maximize sharing of immutable data
  - 3. avoid shared data



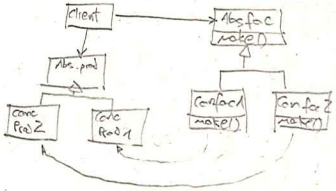


- Design patterns
- Architectural styles
  - pipe-and-filter style
    - components (filters) & connectors (pipes)
    - data flow
    - example of stock
    - StreamIt
    - (split, join, composition)
    - Fusion, Fission
- Procedural coupling
  - function calls
  - 1. move code (duplicate)
  - 2. event based style
    - Observer pattern
    - pro / cons
  - 3. restricting calls
    - tier/layered architecture
  - MVC (model view controller)
    - architecture (using events)

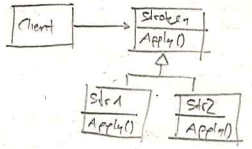


- Class coupling
  - what causes it
    - inheritance, type declarations, object allocation
  - 1. replace inheritance with aggregation
  - 2. user interfaces
  - 3. delegate allocation
    - Abstract Factory pattern

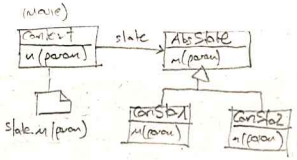
- subtyping
- aggregation
- delegation



- ADAPTATION
  - Change
- Parameterization
  - manipulated value
    - variable values
  - types
    - generics
  - data structure
    - interface and factories instead of classes
  - algorithms
    - pass function as argument
    - delegates (= closures/funct pointers/agents/lambda functions)
    - Strategy pattern
- Specialization
  - overriding
  - dynamic method binding
    - (like case distinction)
    - static vs dynamic (pro/cons)
    - adaptation, aggregation

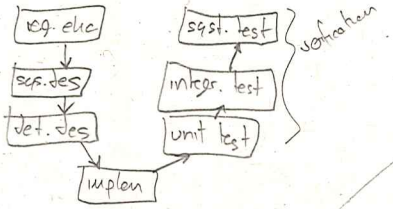


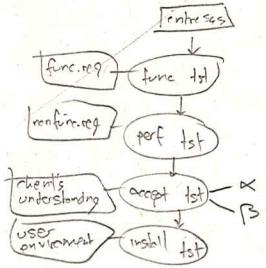
- State pattern
- double invocation
- Visitor pattern



5. TESTING

- Bugs
  - wrong assumptions, design/coding errors
- Fault
  - avoidance (static analysis/development, verification)
  - detection (dynamic, testing)
  - tolerance
- Testing
  - goals / limitations
  - Test Harness
    - Test Driver, UUT (unit under test), Test stub → partial/temporary implementation of component used by UUT
- TEST STAGES:
  - Unit test
    - parameterized, oracles
    - (re)execution
    - when, automatically
    - regression testing
    - 8 rules
  - Integration test
    - big-bang/bottom-up/top-down
  - System test
    - functional (black box, test cases)
    - performance
    - acceptance (alpha/beta)
    - installation
  - Independent testing
    - external team (not programmer)
- TEST STRATEGIES:
  - 1. exhaustive testing
  - 2. random testing
  - 3. functional testing
    - cover all requirements knowledge
    - black box
  - 4. structural testing
    - design knowledge
    - focus on cover all the code
    - white box (unit test)





- FUNCTIONAL TESTING
  - Partition testing
    - divide inputs
    - equivalence classes
    - test cases for each
  - Select representative values
    - below, within, above
  - boundary testing → overflow, comparison, iter #, no empty check
  - Combinatorial testing
    - semantic constraint
    - combinatorial selection
    - random selection
    - (independence of value)
  - Pairwise-Combinations Testing (PCT)
    - interaction of variables
    - complexity

n params, d values : #test cases $O(\log n + d^2)$

- STRUCTURAL TESTING
  - Control flow testing
  - Basic blocks
  - CGF (control flow graph)
  - Test coverage (CF testing)
    - statement coverage
    - branch coverage
    - path coverage (not possible)
    - loop coverage (0,1,+)

path: $n_1, n_2, \ldots, n_k$ (entry, exit), edge (n_i, n_{i+1}, c) in CFG

  - Advanced topics of CF testing
    - Method calls
    - dynamically bound
    - (un/documented) exceptions
    - un/checked exceptions

checked: outside p: user input, DB, network, miss. files...
unchecked: defects in p: args, null-ptr, /0,...

  - Data flow testing
    - dependency of data
    - variable definition/use D/U
    - definition clear paths
    - D-U pairs (definition/use) (d,u)
    - DU pairs coverage
  - reaching definition analysis
    - algorithm → see how it works
    - heap structures
  - Interpreting coverage
    - experimental evaluation (seed errors and see results)

PART 2 - PROGRAM ANALYSIS - M. VECHEV

1. SYMBOLIC EXECUTION

- Explore many program executions
  - under approx, false neg (under/over)
- Idea
  - associate symbolic value X_0
  - instead of concrete value $x = X_0$
  - obtain constant formula to query
- SMT solver (satisfiable modulo theory)
  - find satisfying assignment
  - challenges
    - performance
    - decidable logical fragments (support as many as possible)
    - large formulas
    - domain specific ophmizatons
  - optimizations
    - caching (weaker/stronger formula)
- Constraints
  - linear / not linear / functions $5x+6<100$ $x.y+12<29$ $f(x)<30$
- Theories
  - Equality / Peano / Presburger / Integers...
  - Equality / Integers formulas
- Technically
  - Symbolic store (identity, commutativity, transitivity; functions, predicates)
    - initialized, updated
    - $\sigma_s \in \text{Symstore} = \text{Var} \to \text{Sym}$ es $\sigma_s : x \to X_0$
  - Path constraint (pct)
    - start true
    - history of taken branches
    - evaluates conditionals
    - unsatisfiable -> stop exploration

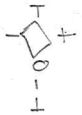
    pct = $x0 > 10 \wedge \ldots$

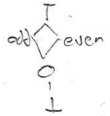
    fork : duplicate state of analysis
- Unbounded loops
  - under approximate (provide loop bound)
  - loop invariant (rarely)
- Concolic Execution (under-approx)
  - due to non-linear constraints
  - concrete store as well
    - drives/steers analysis
    - generate concrete inputs
    - simplifies evaluation
    - external concrete environments (OS, libraries)

es $\sigma : x \to 22$

2. STATIC PROGRAM ANALYSIS (SPA)

- Cool facts
  - what can it do
  - no concrete input
- Abstract interpretation (!!!)
- Step by step
  - 1. select/define abstract domain
  - 2. define transformers
    - abstract semantics for (language x domain)
    - prove soundness
  - 3. iterate transformers until fp
    - start with least abstract element
    - see examples
  - 4. check property
- Fixed Point (fp)
  - over approximation of program
  - not termination problem
- Domains
  - sign
  - parity
  - interval
  - octagon / polyhedral
    - e.g. Apron
  - Properties
    - relational
    - (infinite) height
    - precise enough
    - constraints





- Transformers
  - effect of statement over abstract domain
  - defined per programming language (PL) and domain
  - correctness (/unsound)
  - precise
    - sacrifice for performance
  - best transformer (sometimes impossible)
- Operators
  - split
  - join
    - least upper bound (lub) $\sqcup$
  - widen ∇
- Abstract vs Concrete

3. MATH BACKGROUND FOR ANALYSIS

- Structures types:
- Posets $(L, \sqsubseteq)$ — partial order: bin rel $\sqsubseteq \subseteq L \times L$, L set
  - binary relation
  - set + partial order
  - refl, antisym, trans
  - Hasse diagrams
  - Elements
    - least $\bot$
    - greatest $\top$
  - Bounds
    - upper $\cup$
    - lower ℓ
    - least upper (lub) $\bigsqcup Y$ $\sqsubseteq \cup$
    - greatest lower (glb) $\bigsqcap Y$ $\exists \ell$
    - (non-existence)

$$\begin{cases} \forall p \in L: & p \sqsubseteq p \\ \forall p,q \in L: & (p \sqsubseteq q \wedge q \sqsubseteq p) \Rightarrow p = q \\ \forall p,q,r \in L: & (p \sqsubseteq q \wedge q \sqsubseteq r) \Rightarrow p \sqsubseteq r \end{cases}$$

$p \sqsubseteq q$ "p is more precise" (represents fewer concrete states) "q is more abstract"

$\forall p \in L: \ \bot \sqsubseteq p \sqsubseteq \top$ (may not exist, but if yes → unique)

Y subset: $\forall p \in Y: p \sqsubseteq \cup : \ell \sqsubseteq p$

join / meet

- Complete Lattices $(L, \sqsubseteq, \bigsqcup)$ — es $(\wp(L), \subseteq, \cup, \cap)$
  - poset with lub, glb
  - set of traces $[\![P]\!]$ = reachable statements
- Functions
  - monotone $\forall a, b \in A: a \sqsubseteq b \Rightarrow f(a) \leq f(b)$ $(A, \sqsubseteq), (B, \leq)$
  - iterates $f^0(a), f^1(a), f^2(a), \ldots$ $f^{n+1}(a) = f(f^n(a))$, $f^0(a) = a$, $a \in \bot$
  - algorithm for lfp
- Fixed Point (fix) $f(x) = x$
  - post-fixedpoint (red) $f(x) \sqsubseteq x$
  - pre-fixedpoint $f(x) \sqsupseteq x$
  - least (lfp) $(\mathrm{lfp}^{\sqsubseteq} f)$ — weak, precise
  - Tarski theorem $\forall a \in L: a = f(a) \Rightarrow (\mathrm{lfp}^{\sqsubseteq} f) \sqsubseteq a$ — complete lattice & f monotone → $(\mathrm{lfp}^{\sqsubseteq} f)$ exists
- Approximating functions
  - over approximate
  - SPA function 3 steps
  - functions
    - abstraction
    - concretization
  - Galois Connection
  - Approximation
    - least / most precise — least: $F^\sharp(\bot) = \top$, best (=)
    - best transformer (some behaviour, bit implementation)
- Key theorem
  - 1 $\alpha(\mathrm{lfp}(F)) \sqsubseteq_A \mathrm{lfp}(F^\sharp)$ → requirements: if not: galois
  - automatic computation of lfp
  - 2 $\mathrm{lfp}(F) \sqsubseteq_A \gamma(\mathrm{lfp}(F^\sharp))$
  - F# → approximates F $: A \to A$

$F(\gamma(z)) \sqsubseteq_C \gamma(F^\sharp(z))$

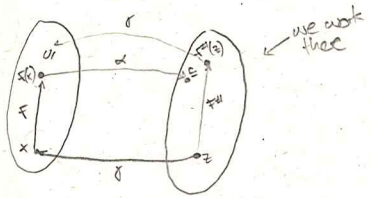
$\mathrm{lfp}^{\sqsubseteq} f = \bigsqcap \mathrm{Red}(f) \in \mathrm{fix}(f)$

$\mathrm{lfp}^{\sqsubseteq} f = f^n(\bot)$ when n stabilizes

approx: $\forall x \in C: F(x) \sqsubseteq_C F^\sharp(x)$ in general $(F, F^\sharp: C \to C)$

$[\![P]\!]$ = reachable states of program P
$F(S) = I_0 \cup \{c' \mid c \in S \wedge c \to c'\}$

$F([\![P]\!]) = [\![P]\!]$



we work here

$\forall z \in A: \alpha(F(\gamma(z))) \sqsubseteq_A F^\sharp(z)$

4. INTERVAL ANALYSIS

lattice	f approx #
$\sqcup$	for lfp
$\bot$	tarski
$\top$	

- Cheat slide
- Set of traces
  - reachable states
  - state properties (assertions)
- Abstraction
  - lose relationship / multiplicity
- Concretization
- Interval domain
  - definition $(L^i, \sqsubseteq, \sqcup, \sqcap, \bot, \top = [-\infty, \infty])$
    - set
    - contains $\sqsubseteq$
    - join / meet $\sqcup$ $\sqcap$
- Applied
  - label / variables $Lab \to (Var \to L^i)$
  - abstraction $\alpha: \wp(\Sigma) \to (Lab \to Var \to L^i)$
  - concretization $\gamma:$ ←
- F
  - approximation (Fi) $(Lab \to Var \to (i)) \to (i)$ $\alpha(\mathrm{lfp}(F)) \sqsubseteq \mathrm{lfp}(F^i)$
  - best transformer $\alpha \cdot F \cdot \gamma$
  - definition
  - fp iteration

$$F^i(m)\ell = \begin{cases} \lambda v. \top & \ell \text{ initial} \\ \bigsqcup_{(\ell', \mathrm{action}, \ell)} [\![\mathrm{action}]\!]_i(m(\ell')) \end{cases}$$

- $[\![$ Action $]\!]$
  - abstract semantics
  - edge in control flow graph (CFG)
  - transitions
  - assign AExp $[\![x := a]\!]_i(m) = m[x \to v]$, $\langle a, m \rangle \Downarrow_i v \in L^i$
    - eval, constant, operators
  - BExp
    - comparison definition $[a,b] \leq [c,d] = ([a,b] \sqcap [-\infty, d], [a, \infty] \sqcap [c,d])$
    - eval
  - skip
- Iteration
  - non termination
  - widening for intervals $\nabla: L \times L \to L$ $[a,b] \nabla [c,d] = [e,f]$ $e = (c < a)\ ?\ -\infty : a$ $f = (d > b)\ ?\ \infty : b$
    - stabilization
    - independence
    - non commutative
    - computation
- Chaotic (async) iteration
  - parallel
  - worklist / influence
- ~~algorithm~~

$(\wp(\Sigma), \subseteq, \cup, \cap, \varnothing, \Sigma)$
$\Sigma = \mathrm{lab} \times \mathrm{store}$

5. POINTER ANALYSIS (not zusammfass)

- pointer / alias
  - heap manipulation
- Programming language
- Concrete store
- Aliases
  - points-to-par $(\vec{p}, \vec{A}), (r, A)$
- May point analysis
  - abstraction
    - allocation sites
    - heap, fixed partition
  - abstract object AS1 $AbsObj = \{\ell \mid stmt(\ell) \text{ is } (p := new \ldots)\}$
  - calling context
- Flow sensitive
  - respect control flow
  - 1. Define domain
  - 2. Define abstraction
    - abstract $\alpha : \wp(\Sigma) \longrightarrow$
    - concretize $\gamma : \longleftarrow$
  - 3. Define abstract Transformers
    - compare
    - create obj
    - assign pointer
    - ptr heap store / load
    - example (in the concrete)
    - (weak updates)
    - structure CE
- Notes
  - handling NULL always present
    - segfault / exception
    - error stop
- Flow insensitive
  - all execution orders
  - concurrency
    - e.g. Soot
  - Abstract domain
    - ignore control flow
    - output
- Alias analysis
  - trivial

points-to(ρ)

$p := new Object^{\ell} T$
$p := q$
$p.f := a$
$x := p.f$
$p.f := q$
$p := q.f$

Obj_s
$PtrVal = Obj_s \cup \{null\}$
$\rho \in PamEnv = Var \to Z$
$re \; PtrEnv = PtrVar \to PtrVal$
$h \in Heap = Obj_s \to (Field \to \{PtrVal \cup Z\})$

store: $\sigma = \langle \rho, r, h \rangle \in Store$

$Labs \to ((PtrVar \to \wp(AbsObj)) \times (AbsObj \times Field \to \wp(AbsObj)))$

– "no integer variables"

$p = q$
$p := new Obj^{\ell} T$
$p := q$
$p.f := q$
$p := q.f$
⚠ phs union, weak updates

start $\rho \to \emptyset$

$\llbracket \cdot \rrbracket : pointer \to (u)\ell = \begin{cases} T & \text{if } \ell \text{ initial} \\ \llbracket action \rrbracket (u(\ell')) & \text{otherwise} \end{cases}$
$(\ell', action, \ell)$

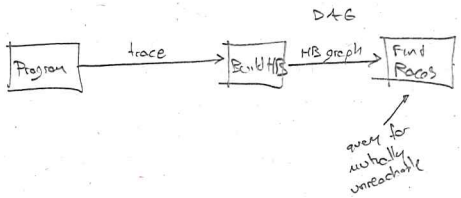
~~Labs~~ $\to (\ldots)$

$:: (Lab \to A) \to (Lab \to A)$

6. RACE DETECTION ANALYSIS

- Specifications
  - Kind of dynamic analysis
  - Concurrency bugs
    - highly asynchronous
  - used in event-driven apps
    - web pages, android
    - user interaction, network requests, ...
- Non determinism
  - asynchrony + shared memory
- Data Race (DR)
  - program state
  - detection
    - outgoing transitions, different threads
    - same memory location
    - one is write
  - naïve algorithm
    - enumeration reachable states
    - doesn't scale
  - in practice
    - ignore states
    - weaken guarantee
    - user defined synchronization
  - Types
    - harmless / harmful
    - synchronization
- Modern Dynamic DR detection 5 steps.
  - 1. Define memory location
    - C like, functions are "locations", JS
  - 2. Happens Before (HB) model
    - poset $(A, \preceq)$
    - define contents and ordering
  - 3. detection algorithm
    - read/writes synchronization
    - no extra false races
    - space / time blowup
    - massive number event handlers
  - 4. remove harmless races
  - 5. implement
    - evaluate performance / precision
- Flow
  - program, trace, HB, dag, find races
- Computing races
  - query if ordered
    - non reachability
    - trace example
- Methods $a \overset{?}{\preceq} b$
  - BFS $O(N)$
  - vector clocks $O(1)$ query (who can reach that node)

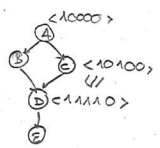
precomp $O(MN)$



query $a \preceq b$

report race: - a,b not reachable
- same mem loc
- one write

no false neg → sound
pos → complete



N nodes
M edges

$vc\text{-}C(B) = 0 \wedge vc\text{-}B(C) = 0$

space explosion

- chain decomposition c ∈ Nodes → Chain IDs C = #chains
  - computation + asymptotic efficiency
  - optimal version
  - greedy version

0. INTRODUCTION

- Turing, halting problem
  - automatic analyzer
  - decide property (undecidable)
  - yes / no (unknown)
- Program Analysis
  - program behaviours / universe
  - over approx (static analysis)
  - under approx (dynamic analysis)
  - over/under (symbolic exec)